LLNL-TH-517351

LAWRENCE LIVERMORE NATIONAL LABORATORY

Target Visualization at the National Ignition Facility

D. Potter

November 30, 2011

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Target Visualization
at the National Ignition Facility

By

DANIEL ABRAHAM POTTER
B.S. (California State University Sacramento) 2006

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Kwan-Liu Ma

Ken Joy

Nelson Max

Committee in Charge

2011

This work performed under the auspices of the U.S. Department of Energy
by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

To my wife

for her support during my studies.

Contents

List of Figures

List of Tables

Abstract of the Thesis

Target Visualization
at the National Ignition Facility

As the National Ignition Facility continues its campaign to achieve ignition, new methods and tools will be required to measure the quality of the targets used to achieve this goal. Techniques have been developed to measure target surface features using a phase-shifting diffraction interferometer and Leica Microsystems confocal microscope. Using these techniques we are able to produce a detailed view of the shell surface, which in turn allows us to refine target manufacturing and cleaning processes. However, the volume of data produced limits the methods by which this data can be effectively viewed by a user. This paper introduces an image-based visualization system for data exploration of target shells at the National Ignition Facility (NIF) at Lawrence Livermore National Laboratory. It aims to combine multiple image sets into a single visualization to provide a method of navigating the data in ways that are not possible with existing tools.

Acknowledgments

I would like to thank my colleagues for their interest during this project. Thanks to Laura Kegelmeyer for keeping her ears open for a potential project for me which led to this work, also for her encouragement to present this work at the IAEA 8th technical meeting. Thanks to Nick Antipa and Alan Conder for the opportunity to create this software and for providing valuable input and enthusiastic responses to each new version of the software. Thanks to Michael Johnson for helping me with the initial concepts and for teaching me everything I needed to know about spherical trigonometry. Thanks to my group leader Jerry Krammen, for taking an interest in the project and providing input and ideas. Thank you to the lab for providing financial support to me in pursuit of my degree.

Special thanks to my review committee, Dr Kwan-Liu Ma, Dr Ken Joy, and Dr Nelson Max. Thank you for taking the time to review this work and provide feedback.

Chapter 1

Introduction

This paper introduces an image-based visualization system for data exploration of a spherical object. The system is being designed to visualize target shells for the National Ignition Facility at the Lawrence Livermore National Laboratory. This chapter gives a brief overview of the NIF itself, then describes the specific problems pertaining to target inspection that will be addressed by the software.

1.1 Background on the National Ignition Facility

The National Ignition Facility (NIF) [1] is the world's largest and most energetic laser, and is a premier laser research facility. It has been built with the goal of being the first facility to demonstrate controlled laser-driven nuclear fusion to achieve ignition, a condition where more energy is produced in the reaction than went into the lasers used to create it. It will achieve this goal through a process called inertial confinement fusion (ICF). In ICF, extreme force is applied to a hydrogen-fuel-filled capsule the size of a BB, causing it to implode and drive a fusion reaction. This reaction produces temperature and pressure conditions not unlike those that exist in stars and giant planets. Future experiments on the NIF will lead to advancements in basic science, maintain the reliability and safety of the U.S. nuclear stockpile without full-scale testing, and lead the way to new solutions in America's energy future [2] with the Laser Inertial Fusion Energy (LIFE) project.

ICF is achieved at the NIF indirectly through the use of a hohlraum (German for “empty room”). The fuel capsule, or target, is situated in the middle of a gold can-shaped hohlraum and is sandwiched between two thin sheets of material to hold it in place. A cone of 96 laser beams are fired into each end of the hohlraum, hitting the inner walls of the hohlraum rather than hitting the target directly. This produces high intensity x-rays which are evenly distributed across the surface of the target, causing it to implode. This process is illustrated in Figure 1.1

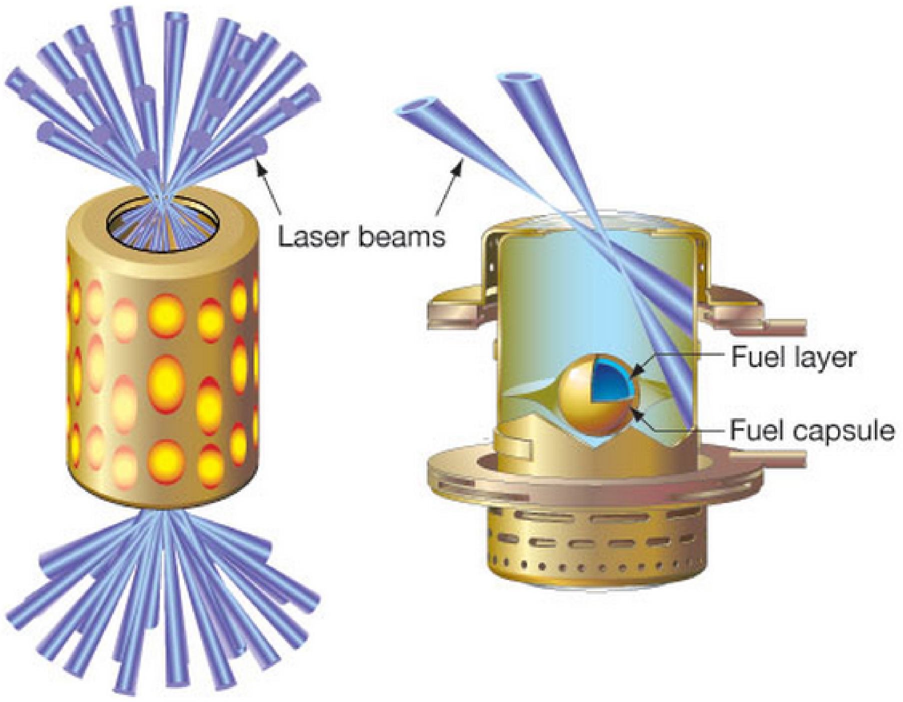


Figure 1.1. Indirect-drive Intertial Confinement Fusion. A target is held inside a hohlraum and is evenly compressed by x-rays emitted when lasers hit the hohlraum walls.

Symmetry is very important in this system. In order to get a fusion reaction, the implosion must be as even as possible across the surface of the target. This means the arrangements of the NIFs 192 lasers must be precisely symmetrical going into either end of the hohlraum. The laser pulses must reach their destination at the exact same time so that the hohlraum is heated evenly. The targets themselves need to be machined within an accuracy of a micrometer. Any unevenness on the surface of the target could cause an uneven implosion, degrading the chances for ignition to occur. To better evaluate the target manufacturing process, and to gain insight on how the shape of a target's surface can affect the performance of a shot, techniques have been developed to map the outer shell of a target using a phase-shifting diffraction interferometer (PSDI) and Leica Microsystems confocal microscope.

1.2 Target Inspection at the NIF

While many aspects of the target surface are monitored, such as the smoothness of the inside of the target and the uniformity of the inner ice layer that divides the target shell and interior gas, this project is mainly concerned with monitoring the outer surface of the target. An ideal target surface is completely smooth at the microscopic level, with no features that would affect the symmetry of an implosion. In reality, the physical scale of these one millimeter wide targets makes it a challenge to produce one without any manufacturing defects, however small they may be.

The techniques and machinery developed to image the outer surface of a target shell allows us to do several things. First, it gives us a baseline state of the shell, that is the state of the shell before or after it was delivered to us (off-site stations exist at our manufacturer's location). It allows us to re-assess the surface of the shell after putting it through a conditioning or cleaning process. Finally it allows for the possibility to orient experiment data taken from the NIFs various in-situ diagnostic systems (that is, equipment that monitors the target while it is in the holhraum and the laser is firing) with actual surface features of the target. This allows us to refine manufacturing and cleaning processes, and can provide insight into the experimental results at the NIF.

As mentioned previously, this mapping is done using a PSDI and confocal microscope. The process of taking the measurements with a PSDI is described in [3]; the process for a confocal microscope is similar. In summary, a target shell is attached to a stage that can rotate about two axis, while keeping the target at the very center of rotation. The microscope takes images of a small portion of the target surface, and the target rotates beneath it through hundreds of stages until the entire surface is imaged. Metadata are recorded at each stage specifying the angles, labeled (θ, ϕ), that the stage has been rotated for each axes. Each instrument produces several hundred images, one or more images at each stage of the process. The dimensions of the images vary depending on the settings of the instrument, an entire target dataset from a single microscope typically takes 2-4 Gigabytes on disk.

Some problems arise from this new capability. The primary problem is that the volume of images produced makes it difficult for any human to visually process the datasets in a reasonable amount of time. Images can be browsed in a file system, but they can only be viewed individually, or at best as a mosaic. There is no way for the user to know the orientation of an image relative to other images in the set using this method. The high number of images the user would have to cycle through would require them to develop a system of remembering what images were of interest, distracting them from their intended goal.

Another problem is aligning multiple datasets of the same target. Multiple datasets are commonly taken of the same target shell at different points in time. For example: one could be taken by the manufacturer using the PSDI system; another could be taken on site when the target is received with the confocal system; and another could be taken after applying a conditioning process to the target. There is no method of securing the target in a specific alignment when imaging it at a workstation, therefore each dataset will have its own coordinate system. There is no easy way to align common features found between multiple datasets because features will not have the same coordinates. Automated image analysis techniques could be used to align features across datasets originating from the same type of microscope, but this capability does not yet exist. It would be even more difficult to match features from the PSDI microscope with features from the Leica microscope, because the images produced by the two instruments are very different.

1.3 Goals for this Project

This visualization software was built with the goal of addressing the unsolved issues in our target inspection process. It defines a database structure to store and retrieve dataset images, and a viewer that arranges and aligns the images of a dataset in a 3D environment that the user is able to rotate and zoom in on. The viewer is capable of displaying entire datasets, and can display multiple datasets at once which the user can control individually. It provides controls and functions built to facilitate the alignment of multiple datasets of the same target, and provides a method of navigating the image data in ways that are not possible with existing tools.

Another goal for the viewer is that it be available over the web. We use a new technology, WebGL, to render 3D graphics directly in the web browser.

1.4 Organization

In chapter 2 we describe the design and implementation of the different modules built for the system. Chapter 3 introduces the functionality and reviews algorithms used by the visualization software. Chapter 4 contains some discussions about the work and possibilities for future work regarding the project.

Chapter 2

System Design

Figure 2.1 shows the architecture of the overall system. It consists of two primary modules. One module which runs on a small cluster of servers loads raw microscope data into a database, the other consists of a web application and web service used to query and display the data. This chapter describes the implementation of these modules.

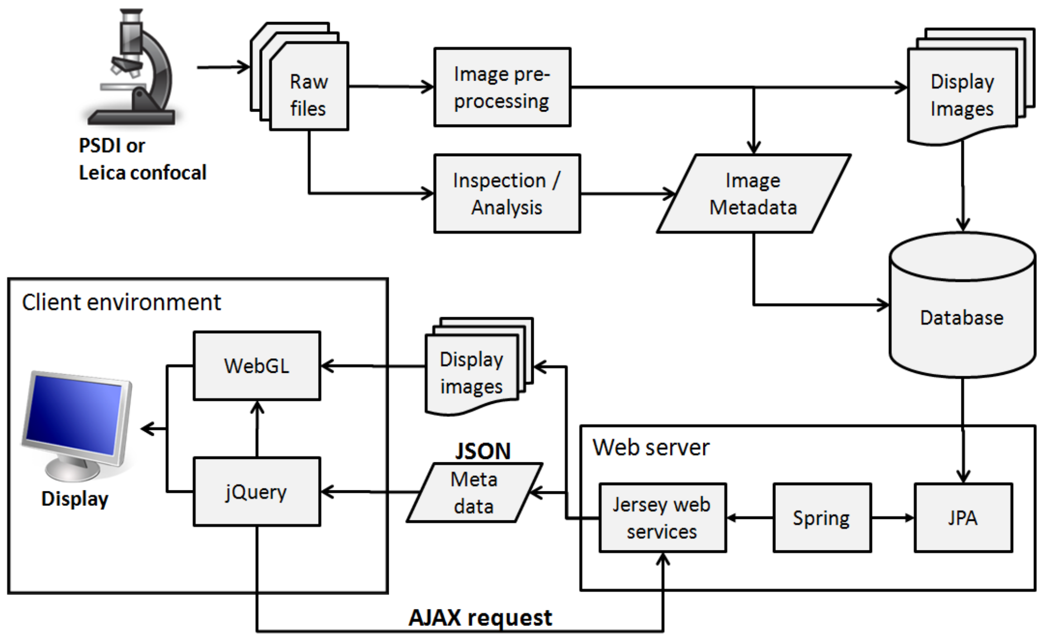


Figure 2.1. High level system architecture.

2.1 Inspection and Data Storage

Before visualizing a dataset we pre-process it and load it into a database for quicker visualization. Raw image data and metadata are collected from the microscopes and uploaded to an Oracle database. Leica data is stored in a proprietary PLU format and PSDI data is stored in an IGOR format. Both formats include floating point arrays representing the intensity and height data of the measurement. Java based file readers have been implemented for both of these formats. For some images, an image analysis program runs which identifies features of interest in the individual images and outputs their locations. Important metadata such as the spherical coordinates of the shell relative to the microscope at the time an image was taken is also stored with every image.

Display images are created from the raw intensity and height data during the upload phase. Images are stored at multiple resolutions so that the lower-resolution images can be downloaded first in the visualization, and higher-resolution images can be downloaded on demand. This reduces the initial wait time for the user. These display images are stored in the PNG image format, and add less than 10% to the storage requirements of the raw data.

Figure 2.2 shows the data model that the raw image data is translated into. The highest level item is an ImageSet, which is composed of multiple Images. The ImageSet setType defines the microscope that the images come from. In this representation, Image contains metadata associated with a single image taken by the microscope. PSDIImage and LeicaImage are extensions of the Image table, and contain special attributes specific to their respective systems. For example, the PSDI images do not have multiple microscope objectives, but do have a concept of hemisphere due to the machinery of the device and the way the images are taken. The DisplayImage table contains actual PNG image data in a BLOB-type column. Each Image will have several display images created for it at varying resolutions. The Detection table is populated by an image analysis program external to this system. It records the (θ, ϕ) locations of features found in each image along with

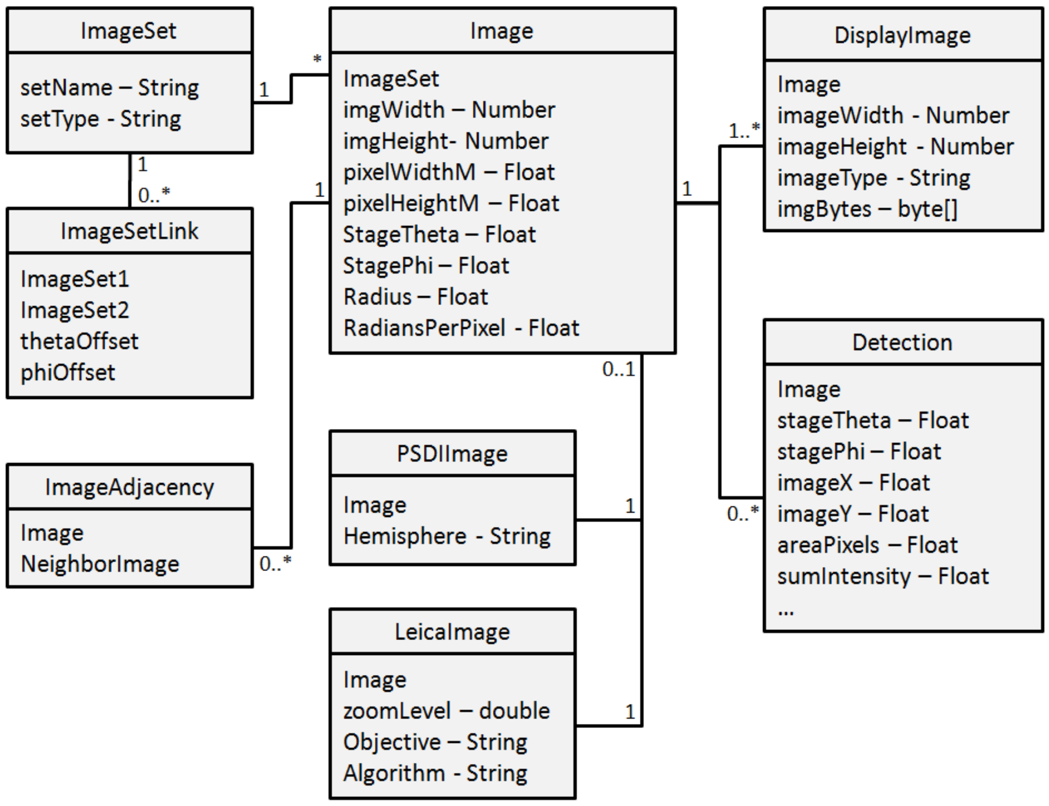


Figure 2.2. Data Model for image sets.

many attributes of those features. This table is linked to the Image table so that Detections can be displayed in the UI.

The ImageAdjacency table keeps track of which Images are neighbors to a specific Image, a neighbor being an image that is physically close enough to another image that some portion of the images overlap. This adjacency information is used for various operations and render modes in the visualization software. During the upload process, adjacency is determined using the algorithm listed in Table 2.1. In this algorithm, each image's (θ, ϕ, r) coordinate representing the center of the image is converted to a 3D Cartesian coordinate and the Cartesian distance between every image pair is calculated. The distance of an image's closest neighbor is saved, and any image within that minimum distance modified by an s factor is considered a neighbor. s can be modified depending on the properties of the image. We assume each image in the set has at least one neighboring image, so this simple method

Table 2.1. Algorithm for determining image neighbors

```
Image[] findNeighbors(image, allImages[], s){
  //first calculate distance between this
  //image and all images in the set
  foreach img in allImages{
    //find cartesian distance between image centers
    dist = getDistance(img, image);
    img.dist = dist;
  }
  var minDist = shortest non-zero distance;
  foreach img in allImages{
    if img != image{
      if img.dist < minDist * s {
        add img to neighbors;
      }
    }
  }
  return neighors;
}
```

makes having to calculate image intersections unnecessary.

2.2 Web Application Architecture

The visualization software is implemented as a web application utilizing javascript libraries on the client side and using a Java based back end. Figure 2.3 illustrates the configuration of the different layers of the application. At the lowest level we have WebGL, which displays 3D content through the browser via HTML 5. Above that there is a custom WebGL wrapper layer which simplifies the WebGL API for the jQuery application layer. The jQuery application layer handles delegating user input to the proper components, querying web services for data, and updating the

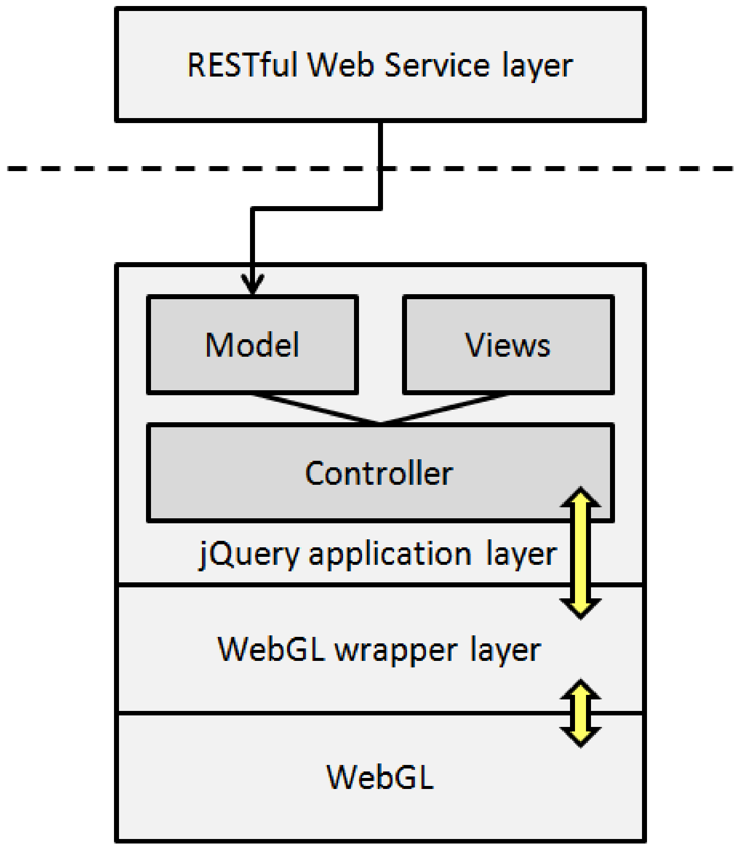


Figure 2.3. High level web application architecture.

view accordingly. The RESTful web service layer is a thin layer in front of the database which exposes a number of RESTful web services for retrieving data.

2.2.1 WebGL layer

The WebGL wrapper layer simplifies communication with WebGL for the application layer by providing some convenience classes and handling basic functions such as managing the render loop. Its functions are listed here:

- **Main framework** - There is a single module which manages the render/update loop. This class is populated with drawable object types, each implementing a draw and update function, which are then iterated through during the draw and update phases of the main loop. This class makes use of the window.requestAnimFrame() function, which allows the browser to determine an appropriate time to allow for another frame of animation to execute. This prevents WebGL loops from continuing to execute when the user has another browser tab active.

- **Navigation** - The navigation model implemented for this project is built using a "scene in hand" metaphor [4], where the object of interest is in the center of view and the user can adjust their perspective relative to the center of the object. In this implementation, the navigation class allows for object rotation and zooming. The minimum and maximum zoom distance is configurable, as is the minimum and maximum zoom and rotation delta values. When the user is fully zoomed out of the scene, maximum zoom and rotation delta values are used when the user changes perspective. If the user is fully zoomed in, minimum zoom and rotation delta values are used. Delta values are scaled linearly between their min and max values based on the user's current zoom level. This scaling gives the users more control in rotating the dataset when they are fully zoomed in to the surface of the object.

  This module also allows navigation to be locked to a specified axis passing through the origin (the assumed center of the object), and has a convenience function which automatically rotates the view to a specified point.

  Other navigation modules could easily replace this one if another type of navigation is needed.

- **Program loading** - The program loader simplifies the interface to loading and configuring vertex and fragment shaders. It handles the loading of shader source code asynchronously, and the binding of all specified uniforms and attributes. An example follows:

```
var tex = ProgramLoader.loadProgram({
        gl : gl,
        fragmentshader : "scripts/app_scripts/shaders/texture_fs.glsl",
        vertexshader : "scripts/app_scripts/shaders/texture_vs.glsl",
        uniforms : ["uSampler","uAlpha","uMVMatrix","uPMatrix"],
        attributes : ["aTexCoord","aVertexPos", "aVertexNormal"]
});
```

In this example, the ProgramLoader accepts a url for the fragment and vertex shader, names of the uniforms and attributes of these shaders, and a reference to the WebGL context. The value that is returned is a jQuery deferred object, which will provide access to the built WebGL program upon resolution.

- **Models** - Model represents a geometry mesh. This class accepts a vertex list, texture list, and index list, and handles generating the necessary WebGL buffers and binding to them during the draw phase. There is also a GeometryGenerator class which generates geometry dynamically and will cache generated geometry if two requests would create the same structure.

- **Picking** - A simple color picking algorithm is implemented. When Models are created, they generate a unique color. When in picking mode, the main framework will set a special override color parameter to the current GLSL program. The scene is rendered to an offscreen buffer, then the pixel underneath the user's cursor is read. The color of the selected pixel is reported back to the application layer. The application layer uses the pixel color to look up in a Map which object corresponded to that color.

2.2.2 Application layer

The application layer is implemented in javascript using jQuery [5] and Javascript MVC [6]. jQuery provides a simplified interface to Javascript that is more cross-browser compatible than pure Javascript is. jQuery also makes it possible to create a page with many distinct and decoupled UI widgets through custom events. In this model, each UI component broadcasts a custom event and arguments associated with the event when appropriate. A master controller listens to these events and is responsible for triggering their response in other UI widgets. Another heavily used feature of jQuery is its ability to make succint Asynchronous XMLHttpRequests (AJAX), and the ability to wait for multiple AJAX requests to finish before moving on to another task using jQuery deferred objects.

JMVC is a library built on top of jQuery that adds some nice features to Javascript. The main feature used in this project is the Class implementation provided by JMVC. Javascript is not a traditional object-oriented language, and does not have object-oriented constructs explicitly built into the language. The JMVC Class adds the object-oriented concept of inheritance to javascript. Classes can be easily defined and have static and instance methods and attributes associated with them. Classes can also be extended into subclasses adding the benefits of polymorphism to the application. In this application, polymorphism is used to simplify the WebGL rendering code between two object types that are mostly similar but differ slightly in how they need to be rendered.

2.2.3 Web Service layer

jQuery asynchronous calls are made to a web service layer implemented in the Java-based Jersey framework [7]. Jersey is a RESTful Web Services library that makes creating web services incredibly easy. Ordinary Java methods are configured using annotations to specify what type of HTTP requests they will respond to, the URL for accessing the method, the parameters accepted by the method, and the format of the response. With a small amount of Jersey configuration, these methods then become accessible simply by visiting a URL.

Jersey can also be configured to automatically convert Java Objects into a JSON formatted string. JSON, a lightweight language-independent data format similar to XML, can be automatically parsed by jQuery into a javascript object. Using Jersey plus jQuery allows server side Java objects to be seamlessly translated into client side Javascript objects. This allows web service methods to look like plain Java methods, and no regard has to be given to making the method compatible over a web interface except for the Jersey configuration.

Also implemented in the server layer is a Java object representation of the database using JPA. JPA allows us to represent database tables as Java objects, where the Object represents a row of the table, and column values are represented by attributes. This allows us to have a consistent representation of our model objects throughout all layers of the application, as illustrated in Figure 2.4.

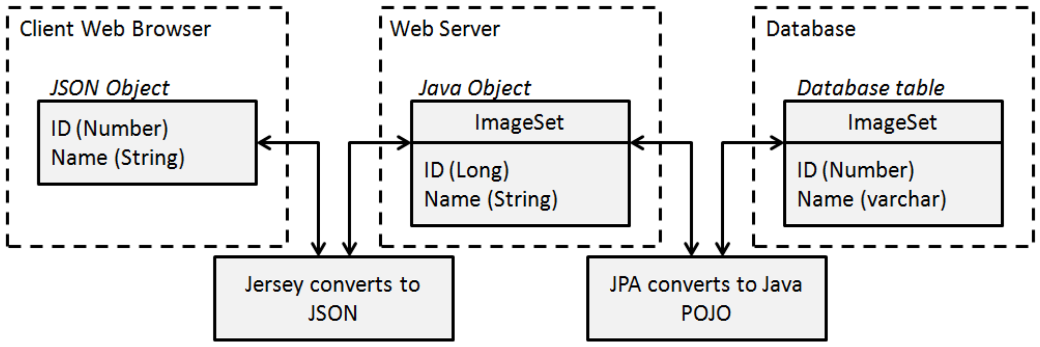


Figure 2.4. Example representation of model objects at all layers of the application. JPA, Jersey, and jQuery make it easy to have a consistent object representation throughout the application while automating the transferring of these objects across layers.

Chapter 3

Visualization Software

This chapter focuses on the web-based visualization software itself. First we look at the visualization software and the features it provides, followed by a look at some of the algorithms used in the visualization. Next we discuss the motivations and merits of our visualization paradigm.

3.1 Functionality of the Application

We now discuss the features implemented in the application in order to address the goals defined in Section 1.3.

Figure 3.1 gives an idea of the utility of this visualization due to the fact that over 200 images are involved. The darker borders of the images in the smaller grayscale dataset make their size obvious. Images in the other dataset are roughly the same size. It would clearly be much harder to view these datasets image by image, or even in groups of images. An advantage to being able to view the entire dataset at once is that it gives users a spatial context for each individual image. Individual images can be right clicked to get image specific options. Higher resolution versions of the image can be requested, as well as switching the image type (height vs intensity) of the image and its neighbors, or of all images in the set.

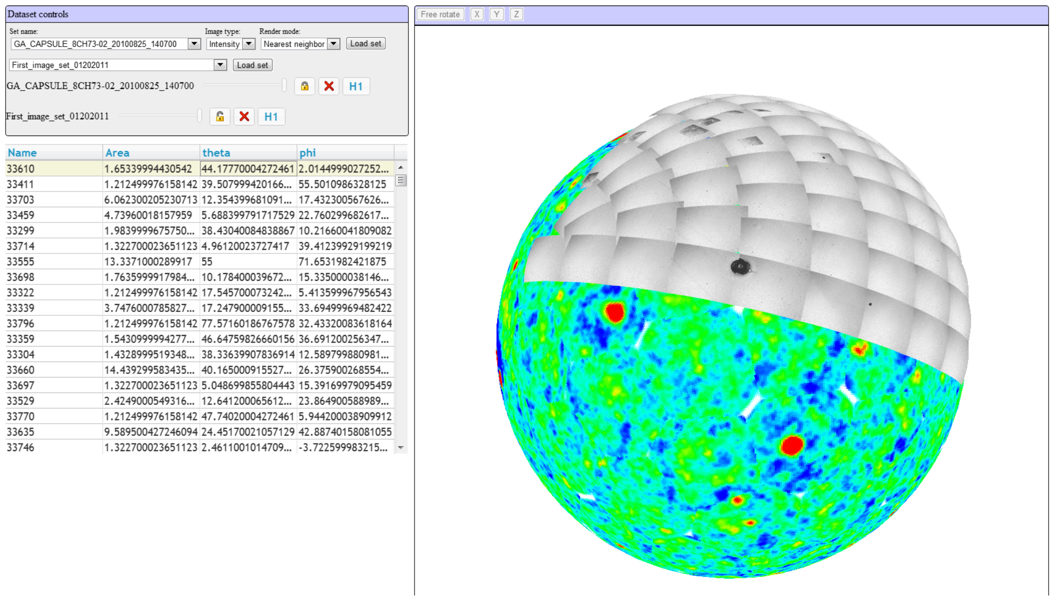
Another feature provided is the listing of features of interest found during the inspection phase mentioned in Section 2.1. Users are able to sort this list by any attribute of interest. Upon clicking a row in the list, the view automatically rotates to the location of the feature that was clicked. This feature is provided by the WebGL navigation module described in Section 2.2.1.

Data set alignment is another task that would be very difficult to perform if viewing data image by image. As mentioned in Section 1.2, the visualization currently supports datasets from two types of microscopes. We have no way of keeping the target orientation consistent between workstations, so features will not automatically line up between datasets.

This visualization provides an easy way to find common features visually. See Figure 3.2 as an example, in which the user aligns the laser-drilled 10-micron wide hole used to insert the fuel fill tube into the capsule. First, the user can lock a dataset from being rotated, allowing them to freely rotate and line up features in the other set. To assist with this process, a slider is provided to control the transparency of each dataset. Second, the user can define an axis of rotation anywhere on the sphere, which locks rotation of the set about that axis. When the user lines up one common feature, they define an axis of rotation in the center of the feature. They then lock one of the sets, and rotate the other about the fixed axis until the rest of the common features line up. The alignment is then saved to the database, so future viewing of the datasets will already be aligned when visualized.

3.2 Visualization techniques

A single PSDI dataset consists of 142 measurement locations, and contains two images per location. Each image consists of a circular region of the surface, called a medallion. Any pixels outside of the image medallion are set to 0 by the time the PSDI system encodes the data into the image array. This makes it convenient for the visualization, as any pixels which are set to zero can be ignored, and are set to be transparent. A further convenience is that the intensity and surface height images in each image pair match each other pixel for pixel, so they can be swapped



Set name: GA_CAPSULE_8CH73-02_20100825_140700
Image type: Intensity
Render mode: Nearest neighbor

First_image_set_01202011

GA_CAPSULE_8CH73-02_20100825_140700

First_image_set_01202011

Name	Area	theta	phi
33610	1.65339994430542	44.17770004272461	2.0144999027252...
33411	1.212499976158142	39.507999420166...	55.5010986328125
33703	6.062300205230713	12.354399681091...	17.432300567626...
33459	4.73960018157959	5.688399791717529	22.760299682617...
33299	1.9839999675750...	38.43040084838867	10.21660041809082
33714	1.322700023651123	4.96120023727417	39.41239929199219
33555	13.3371000289917	55	71.6531982421875
33698	1.7635999917984...	10.178400039672...	15.335000038146...
33322	1.212499976158142	17.545700073242...	5.413599967956543
33339	3.7476000785827...	17.247900009155...	33.69499969482422
33796	1.212499976158142	77.57160186767578	32.43320083618164
33359	1.5430999994277...	46.64759826660156	36.691200256347...
33304	1.4328999519348...	38.33639907836914	12.589799880981...
33660	14.439299583435...	40.165000915527...	26.375900268554...
33697	1.322700023651123	5.048699855804443	15.39169979095459
33529	2.4249000549316...	12.641200065612...	23.864900588989...
33770	1.212499976158142	47.74020004272461	5.944200038909912
33635	9.589500427246094	24.45170021057129	42.88740158081055
33746	1.322700023651123	2.4611001014709...	-3.7225999832215...

Figure 3.1. Screenshot of the application. Dataset controls are shown in the upper left, sites of interest are listed in the lower left. The 3D visualization can be interacted with in the box on the right side. Here two datasets are loaded, one covering an entire capsule and the other covering 1/8th of the capsule.

back and forth in the visualization without needing to change their position.

Leica datasets generally consist of over 300 images, and can contain images at different magnifications. These images are treated mostly the same as PSDI images, except that higher magnification images are ignored in the nearest neighbor rendering mode. Images in this section will be referencing PSDI based images. An example of PSDI images is given in Figure 3.3.

To place images in 3D space, the system uses the (θ, ϕ) location of the stage at the time the image was taken, as well as the radius of the target itself in millimeters. This spherical coordinate is converted to Cartesian coordinates for every image. The image is texture mapped to a geometric primitive which is placed at the converted Cartesian coordinate for the medallion. To make the images face outward, the geometric primitive is rotated about its z-axis by ϕ then about its y-axis by θ. PSDI images have an extra rotation due to how the PSDI measurements are taken. If the image is within the second hemisphere the geometry is rotated again about its z-axis by π.

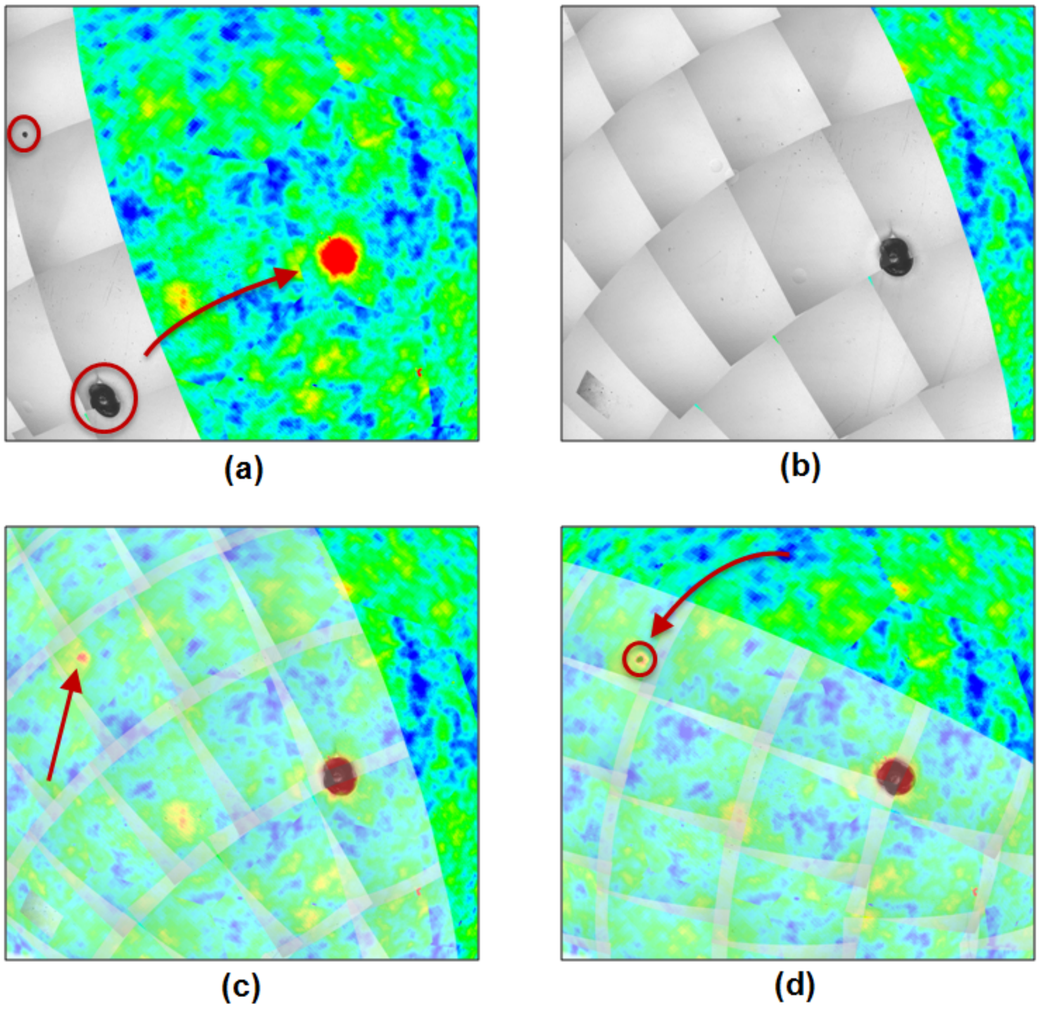


Figure 3.2. An example of aligning two datasets. (a) User finds a common feature, note the smaller circled feature. (b) User locks one dataset and aligns the larger features. (c) User adjusts the transparency to further align the datasets and finds another possible common feature. (d) User rotates about the large common feature to line up the smaller feature from image (a).

Images are mapped to a curved mesh representing the curvature of the target surface. The mesh is generated using the known pixel width/height of the image, as well as the known radians per pixel value of the instrument used to obtain the image. Rendering to the curved geometry as opposed to a flat quad improves the look of the visualization because it makes features line up more accurately and makes user picking for defining an axis of rotation more accurate.

Because an entire full resolution dataset is quite large, the following techniques

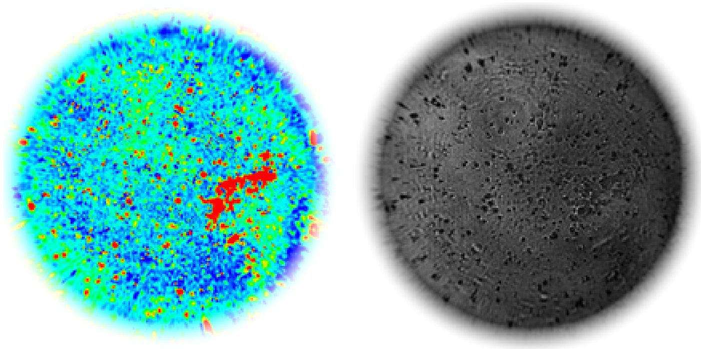


Figure 3.3. A PSDI height map image (left) and corresponding intensity image (right) of a shell surface patch.

are used to minimize wait time for the user. First, only image metadata is loaded initially. The downloading and preparation of textures is handled asynchronously, updating the display as images are loaded. This allows the visualization to stay interactive during the loading process, and results can be displayed as they are ready. The user can interact with parts of the dataset before the complete dataset is available. To minimize dataset download time, lower resolution versions of the dataset images are downloaded initially. The user has the ability to request higher resolution images on demand.

3.2.1 Render modes

Three render modes are provided, full image, nearest neighbor, and blended. These modes are provided to give users flexibility in how they visualize the data, as some may prefer one over the other. Each has trade-offs in the quality of the visualization vs. how much information is displayed. These render modes address the fact that the edges of each image overlap with one or more neighboring images, meaning some areas of the surface target appear in two or more images.

Images for nearest neighbor and blended rendering modes are pre-computed in a OpenGL Shading Language (GLSL) shader program and saved to a texture buffer when the client initially loads a dataset. This prevents us from having to make the calculation every frame, which cannot be done at a high frame rate on most machines. If the user loads in a higher resolution version of an image, the appropriate shader program is rerun for that image and its neighbors using an appropriately sized texture buffer.

3.2.1.1 Full Image

In this default mode, the original images are unchanged. As the user's perspective changes, image geometry is sorted based on its distance to the user's view point. Geometry is then rendered from back to front, ensuring that the image closest to the user's center of view is rendered above its neighboring images. Discontinuity between images is obvious in this mode because image edges pop in and out of occlusion as the user rotates the model. Users may consider this method hard to interact with, as the behavior described may be distracting. An advantage of this render mode is that all portions of the image data are visible to the user, so they can be reassured that no data is lost in the visualization. An example is shown in Figure 3.4.

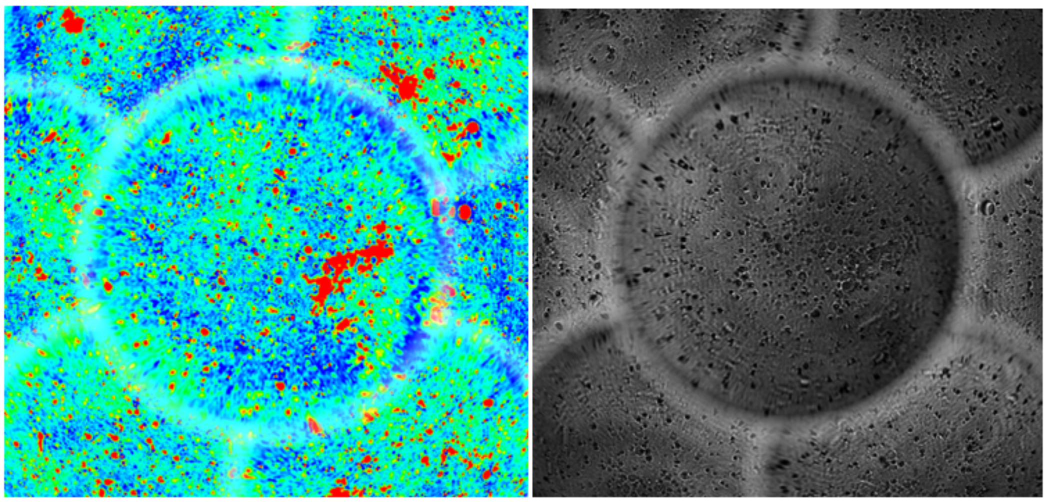


Figure 3.4. Examples of full image rendering mode. As the user rotates the view, neighboring image edges pop in and out of view.

3.2.1.2 Nearest Neighbor

In this mode, the distance between each overlapping pixel and each neighboring image center is calculated. Any pixel that is closer to the center of a neighboring image than it is to the center of its source image is dropped from the source image. This allows for a smoother navigation process, as images will not pop into view over each other as in the full image method. It does drop some information where pixels overlap though, information which users may want to see.

Calculating the distance of the pixel between image centers is done in several steps. First, we note that a texture coordinate of an image ultimately represents a spherical coordinate in the target shells global coordinate system. Therefore, we aim to convert texture coordinates of the image to their corresponding global spherical coordinate as described in [8]. The imaging stage is configured such that the x-axis of the image runs along the θ direction, and the y-axis runs along ϕ. θ runs from 0 to 180 degrees and ϕ runs from 0 to 360 degrees. If we follow along the x-axis from the center of the image, we will run through a pole of the sphere at $\theta = 0$. We imagine the pixel location (p, q) forming a right spherical traingle with this pole and the x-axis of the image. This forms the triangle abc shown in Figure 3.5.

Of course in reality the triangle in the image plane is a right triangle, but the orthographically projected triangle on the sphere is not a right spherical triangle because side a will not be a portion of a great circle. However if we assume the distances X and Y from the center of the image will be much smaller than the radius of the shell, as is the case here, we can approximate the triangle as a right spherical triangle. Using this model, we can conveniently use Napier's rules for right spherical triangles to find the (θ, ϕ) which corresponds to (p, q).

First, a relation between linear image coordinates and pixel coordinates is formed.

$$X = \alpha(.5I_w - p)r$$

$$Y = \alpha(q - .5I_h)r$$

where I_w and I_h are the images width and height in pixels, respectively. p and q are the pixel coordinates of the image. The origin of the pixel coordinates is at the lower right corner of the image by convention of the imaging system. r is the radius of the shell, and α is a fixed radians per pixel value which represents the angular

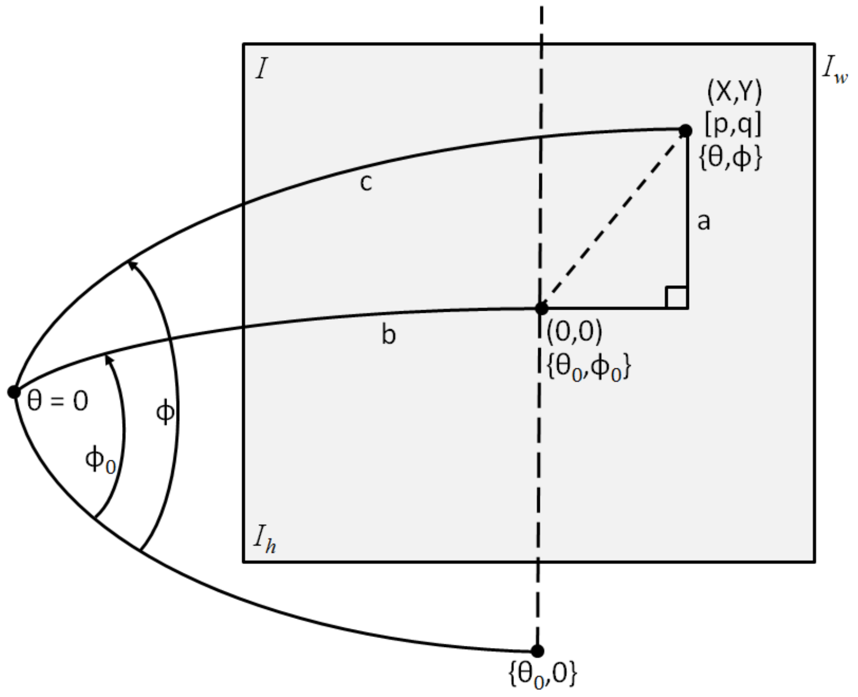


Figure 3.5. We consider the origin of the image to be at $(0, 0)$ in linear image coordinates. The origin of the image position in spherical coordinates also lies at the center of the image. The location we are trying to convert is at the known pixel coordinate (p, q).

magnification of the instrument. Recall that because this is a spherical triangle, side lengths are expressed in angles relative to the center of the sphere. Therefore sides a and b can be estimated as being equal to $\frac{Y}{r}$ and $\frac{X}{r} + \theta_0$ respectively. These values are used in Napier's rules to form the equations below. The equations can be formed using Napier's circle mnemonic device.

$$cos(\theta) = cos(\frac{Y}{r})cos(\theta_0 + \frac{X}{r}) \tag{3.1}$$

$$\begin{aligned} sin(\theta_0 + \frac{X}{r}) &= cotan(\phi - \phi_0)tan(\frac{Y}{r}) \\ tan(\phi - \phi_0) &= \frac{tan(\frac{Y}{r})}{sin(\theta_0 + \frac{X}{r})} \end{aligned} \tag{3.2}$$

where (θ_0, ϕ_0) are the spherical coordinates of the center of the image as recorded by the instruments. Substituting X and Y in equation (3.1) and (3.2) and solving for (θ, ϕ) yields

$$\theta = cos^{-1}(cos(\alpha(q - .5I_h))cos(\theta_0 + \alpha(.5I_w - p))) \tag{3.3}$$

$$\phi = \phi_0 + tan^{-1}(\frac{tan(\alpha(q - .5I_h))}{sin(\theta_0 + \alpha(.5I_w - p))}) \tag{3.4}$$

The resulting spherical coordinate is then converted to a global Cartesian coordinate and its Cartesian distance is compared with the Cartesian origin of the parent medallion, and all neighboring medallions. If the point is closer to a neighboring medallion than it is to the parent medallion, its alpha value is set to 0 for that image, knowing that its place will be taken by a pixel in one of the neighboring images. This algorithm is defined in Table 3.2.1.2.

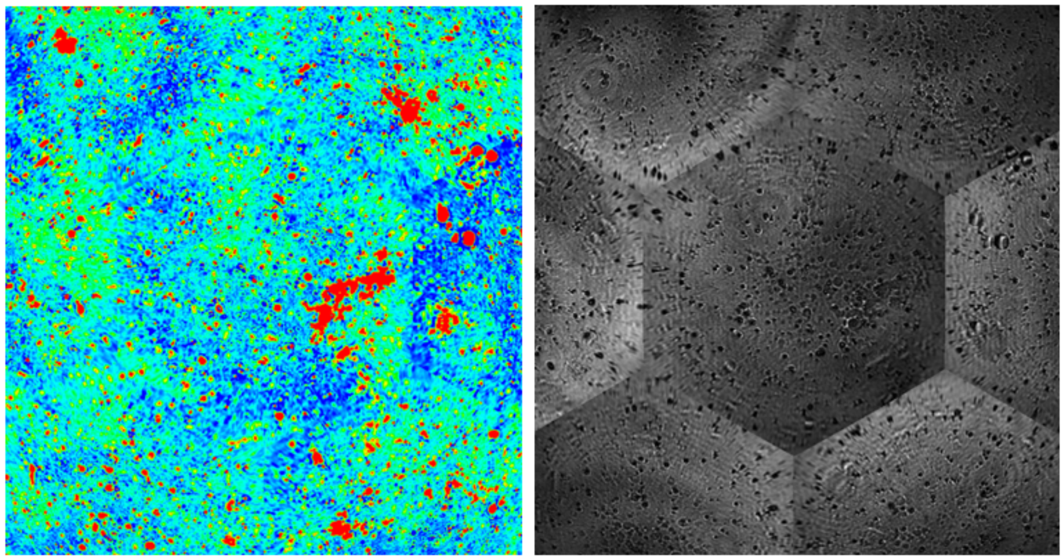


Figure 3.6. Examples of nearest neighbor rendering mode. The user can see more details of other images in this mode, note the area to the right in the left image that was partially occluded in full medallion rendering mode. Also note how pixels near the edge of every image are thrown away in this mode.

3.2.1.3 Blended

This mode goes a step past nearest neighbor and attempts to create a view that is similar but does not drop out important pixels. Important pixels are those in

Table 3.1. Algorithm for nearest neighbor rendering

```
renderNearestNeighbor(){
  //convert the current texture coordinate
  //into a spherical coordinate
  //this uses equations 3.3 and 3.4
  spos = convertToSpherical(imgCenter, texCoord);
  cpos = sphericalToCartesian(spos);
  thisPos = sphericalToCartesian(thisPos);
  dist = distance(cpos, thisPos);
  alpha = 1; //assume this pixel will not be hidden
  foreach neighbor in neighbors{
    ndist = distance(cpos, neighborPos);
    if ndist < dist{
      alpha = 0; //this pixel is closer to a neighbor
      break;
    }
  }
  return texture lookup with alpha set;
}
```

the surface height images which represent a higher elevation. This method is more useful for PSDI images, where height data gets noisier the further away a pixel is from the center of the image, so overlapping pixels will not necessarily agree on height. Pixels in the intensity images do not have a measure of importance. This render mode could be modified to use the height image data pixels as a measure of importance for blending the intensity image pixels.

To achieve this, we must convert the input texture coordinate to a global spherical coordinate as in the nearest neighbor algorithm. Using this coordinate, we calculate what the corresponding pixel location would be for each image, relative to the center coordinate of that image. Once all of the corresponding pixels are

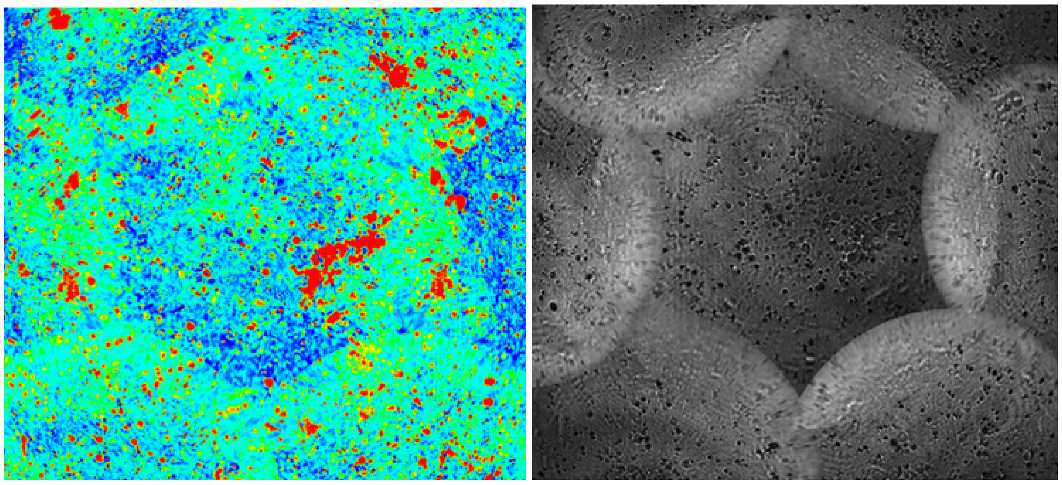


Figure 3.7. Blended mode shows more detail at each individual view point in the surface height images by displaying the color corresponding to the maximum height (shown red here) in overlapping areas. This mode is not as useful when viewing intensity images.

calculated, each neighboring image is sampled at the calculated pixel, and the pixel representing the maximum height is kept for display.

As we did in calculating the spherical coordinate at a pixel, the equations for p and q can also be derived using Napier's rules. In this example, we solve for q using napers rules, then solve for p using a re-arranged form of equation (3.1) where p is defined in terms of Y, and using q to solve for Y.

$$q = \frac{sin^{-1}(sin(\theta)sin(\phi - \phi_0)}{\alpha} + .5I_h$$

$$p = -(\frac{cos^{-1}(\frac{cos(\theta)}{cos(Y)}) - \theta_0}{\alpha}) - .5I_w)$$

An example of blended mode is shown in Figure 3.7.

3.3 Navigation techniques

The navigation class built for this project has some features included to make navigating the scene more user friendly.

- **Controlled rotation** - Users can define an axis of rotation, then toggle navigation to rotate freely, or confine rotation about the defined axis. First the user chooses a location on an image they wish to rotate about. Color picking is used to determine which image was selected, and a radius value is obtained from that image. A vector originating from the mouse location is then intersected with a sphere located at the origin with a radius equal to the selected image. The axis of rotation is defined by this (x, y, z) location.

- **Auto rotation** - The view can automatically rotate to any point of interest. This is used to automatically rotate the view to a detection when the user clicks one in the detection list. First one vector is defined by the location of the selected detection and the center of the sphere. Another vector is defined along the user's viewing axis. The cross product of these two vectors produces a perpendicular vector which we use as our new rotation axis. The arc cosine of the dot product of these vectors will yield the angle between the two vectors. The view is rotated about the new axis by the new angle to bring the selected location into view. The user is also allowed to snap the defined axis of rotation to the X, Y, or Z axis.

- **Smooth navigation** - When zooming or auto rotating, a target destination is defined rather than abruptly jumping to the new location. On each frame, if the current zoom level or rotation is not equal to the target one, they will be gradually updated based on their distance to the target value. This keeps the automatic movements smooth and allows the user to maintain the physical context of the data.

3.4 Motivation

We now make a case for implementing the visualization in the described manner. One could imagine other visualization solutions to the problem, such as stitching all images together in a 2D plane or projection that could be panned and zoomed by the user, or an interactive image viewer that allowed users to view multiple images at a time and provided some specialized navigation controls. These implementations would have a desirable advantage in that they may not require hardware acceleration. We feel that a visualization that simulates the physical appearance of the original object is a superior solution.

Techniques in this area vary depending on the item in question and the method in which the images were taken. Many scientific visualization applications use volume visualization, either through direct volume rendering [9] or an iso-surface extraction method [10], to generate a detailed 3D rendering from a stack of image slices. Examples of these types of datasets include images generated by MRI or CAT scans for medical purposes, or by confocal microscopes for biological studies. These are useful visualization techniques, because they allow the user to see an accurate reconstruction of the object in question, instead of having to visualize it in their head from hundreds of 2D images.

[11] and [12] reconstruct real world objects using sets of range images to build a complete surface mesh of the object. Techniques like these are useful for creating accurate 3D representations of an object's surface, as long as the object can be conveniently imaged. These techniques are not generally used for visualization software, but are useful in entertainment industries. The end goal is typically to produce a high quality 3D model for professional use in the video game or movie industry for example.

Instead of reconstructing an individual object, [13] and [14] reconstruct entire real-world scenes composed of individual photographs from image databases into a 3D navigable environment by positioning images according to the angle and location that they were photographed from. This allows users to view each image in relation to images that were taken nearby, giving them a better idea of what it is like to be at the scene. Similarly, Google Earth, a GIS visualization system which is frequently used as an image-visualization platform, is used to provide spatial context for image visualization. [15] describes a system using Google Earth for browsing news related images which are displayed on a map at the location of the city where the story originated from.

Clearly a spatial reconstruction of the subject matter can be more valuable to the user than viewing subsets of the data individually. This idea is used as motivation for our visualization. Since the images obtained by our microscopes do contain range data, we may be able to use techniques from [11], [12], or [10] to create a more accurate reconstruction of the target surface. However, a simple image representation of the data does suffice to meet the goals of this project. Also the geometry of the features is minuscule relative to the size of the capsule, so these techniques may not produce anything useful.

Since we are using images, this system can be thought of as an image browsing system. Similar to [13] and [14], it accurately places and orients images such that they are aligned properly with their neighboring images. Viewing a dataset in this way allows users to associate a spatial context with each image because they can see its surrounding images. They can more easily identify features and patterns that span multiple images.

3.4.1 Inspiration from Google Earth

It is likely that the most well known software that is similar to the one described is Google Earth, a 3D GIS visualization tool developed by Google and provided for free. Google Earth runs on a user's desktop or as an embedded web application, and is powered by a local graphics card. It is very similar functionally to the software for target visualization defined here. They both use a "scene in hand" visualization metaphor of spherical objects. They have similar use cases in the viewing functions users would want to perform. They allow the user to rotate the object, zoom in, jump to, or mark a location of interest. They also both use performance boosting techniques such as graphics acceleration and level of detail for images.

Google Earth itself is highly customizable, and has become a visualization platform for GIS related applications. Using its custom Keyhole Markup Language (KML) xml definition, users can define graphic overlays such as points, lines, polygons, images, or 3D models to be displayed at specific locations on the map. This system gives the user a lot of flexibility in associating various information with points of interest on the map, and has been used in various GIS based visualizations.

Using Google Earth as a platform for this visualization project seemed promising and was investigated. It was ultimately determined that it was not suited for the visualization of target shell data. For one, it would be difficult or impossible to define new modes of operation in Google Earth. The controls required for dataset alignment need a way to manipulate two datasets independently of each other. Secondly, Google Earth's KML files did not give enough control over the presentation of the user interface, so many GIS specific features would be displayed while inspecting a target shell. This would be a source of confusion for users.

Chapter 4

Discussion

4.1 Why a Web Application?

The user interface was implemented as a web-based application because the centralized nature of such applications makes them easier to develop, test, deploy, and support. However until recently, the sort of 3D visualization we wanted would not have been possible in a web application without making some sacrifices. Other technologies could have been used, such as an embedded java applet running JOGL or another java based 3D framework, or a Silverlight / Adobe Flash solution, or a browser plug-in that enabled 3D content. All of these technologies sacrifice one or more of the benefits of having a web-based application. Java 3D frameworks are not always compatible across operating systems, which makes it harder to develop and support software for multiple platforms. Silverlight and Flash are capable of 3D, but are underpowered, and use proprietary languages that most developers at NIF are not familiar with. Other browser based plug-ins are not well supported and have never reached maturity, and have issues with working across browsers.

The current state of 3D content on the web is summarized in [16]. This article discusses a new technology that has recently been incorporated into modern browsers called WebGL. This framework allows rendering of 3D content directly in the browser without the need of any plug-ins. The user does not need to perform any setup in order for it to work as long as they are using a supported browser.

As of this writing supported browsers are Chrome and Firefox 4, with support in upcoming versions of Safari coming.

WebGL is a promising framework for the future of web-based 3D content. First, unlike most of the other technologies previously mentioned, WebGL utilizes local graphics hardware to deliver performance the other frameworks cannot match. This also gives developers access to run shader programs directly on the GPU, allowing them to implement modern GPU-based algorithms. Second, while it is a new specification for web browsers, it is based on the well known and widely used OpenGL framework. The API is based on OpenGL ES 2.0, a subset of the OpenGL 2.0 library, used primarily for 3D rendering in mobile devices such as the iphone and android platforms. WebGL also has an active community, with many budding new frameworks being created to ease its use, such as [17–19] and research on using it for web-based visualizations [20].

As previously mentioned, non 3D aspects of the application, that is, all other portions of the user interface, are managed using the javascript library jQuery, and other libraries built around it. Interface elements are populated through RESTful web services using Java's Jersey framework. We have found that javascript based frameworks, along with RESTful web services, make the UI easier to develop and test because the user interface layer and service layer can be built and tested individually; other frameworks tend to couple these components. Javascript frameworks also lead to better performance and a wider range of possibilities in the functionality of the application, as opposed to a component based java framework such as Java Server Faces.

The viewer was originally implemented as a Java based web-start application using Java3D. This would enable the application to be launched from the web but ultimately run on a user's desktop. A web application was built in favor of this initial prototype because it was easier to get the application working across operating systems. The future of Java3D is also questionable; at the beginning of this project it looked as if Java3D would be incorporated into JavaFX, which is being promoted as a method of creating rich Internet applications similar to Adobe Flash. JavaFX seems to not be getting as much attention since Oracle purchased Sun however, so its future is questionable as well. WebGL and HTML 5 were interesting new solutions to explore so the visualization software was re-implemented as a web application.

4.2 User response

This application has been in use at NIF since its first prototype release. It has proven to be a useful tool for the purposes described in the introduction. Early on the tool was used to figure out the alignment of a target shell after a shot was taken by lining up features in the images with images from NIF's target chamber diagnostic tools. More recently the tool has been used to check the efficacy of cleaning techniques on the shells. A pre- and post-cleaning dataset will be taken, then the two will be compared in the viewer. Dataset alignment controls are used first to align the two datasets, transparency controls are then used to compare the pre-cleaning results with the post cleaning results.

4.3 Future work

Future work on this project could involve adding to the functionality of the application and improving the quality of the visualization. Removing image seams to make the surface continuous could be a desirable feature in future versions of the software. The current visualization relies on the accuracy of the inspection machinery for images to line up. Images normally line up quite well, but in some cases it is obvious that they are off. This can be caused by the limits of precision in the equipment or user error. An image stitching algorithm could overcome these issues. [21], which deals with the combination of spherical images looks especially interesting. A more robust technique for determining pixel visibility using ray sphere intersection suggested during the review of this work may also be explored. It may also be interesting to add lighting support in the visualization, and use the image height data to implement bump mapping on the image textures.

Because the software is actively being used, users have defined several features that they would like implemented in the future. One desired feature is to create a detailed 3D view of a user-specified region of an image. This could be done by having the user specify a region of the shell for the detail view. Height data could then be used to create a more detailed 3D mesh of the user-specified region of the shell. Another feature is to have predefined viewpoints that the user can view the target shell from. These viewpoints would represent the NIF diagnostics that sit in the target chamber and gather data on the target as the laser is firing. Having predefined viewpoints would allow users to more easily see what diagnostics line up with surface features.

4.4 Conclusion

The National Ignition Facility is a one-of-a-kind facility with one-of-a-kind challenges. The software described here was developed in order for NIF scientists to better view image based datasets of NIF target shells. The software will make it easier for users to get an overall picture of the datasets, and allow them to gain insights that would be difficult or impossible without it. This will be a valuable tool in evaluating the quality of NIF targets, which is an important element for achieving ignition.

Appendix A

code listings

A.1 Nearest Neighbor Rendering source code

```
varying vec2 vTextureCoord;
uniform sampler2D uSampler;
uniform float uAlpha;
uniform vec3 imgCenter;
uniform vec3 neighborCenters[8];
uniform int numNeighbors;
uniform int hemisphere;
//in - centerCoord is a spherical coordinate in radians
//returns the spherical coordinate at texCoord in radians
vec3 convertToSpherical(in vec3 centerCoord, in vec2 texCoord){
   float cTheta = centerCoord.x;
   float cPhi = centerCoord.y;
   float cr = centerCoord.z;
   float rpp = .00077;
   float valA = rpp * ((1.0-texCoord.y)*1024.0 - 512.0);
   float valB = cTheta + rpp * (512.0 - texCoord.x*1024.0);

   float theta = acos(cos(valA) * cos(valB));
   float temp = tan(valA) / sin(valB);
   float phi = cPhi + atan(temp);

   vec3 ret = vec3(theta, phi, cr);
   return ret;
}

//in - a spherical coordinate in radians
//returns a cartesian coordinate
vec3 sphericalToCartesian(in vec3 coord){
   float theta = coord.x;
   float phi = coord.y;
   float r = coord.z;
   float x = r*sin(theta)*cos(phi);
   float y = r*sin(theta)*sin(phi);
   float z = r*cos(theta);
   vec3 c = vec3(x,y,z);
   return c;
}

void main(void) {
   vec2 texCoord = vTextureCoord.st;
   if (hemisphere == 2){
       //mirror the coordinates for the texture lookup,
       //but not for the final texture mapping
       texCoord.s = 1.0-vTextureCoord.s;
       texCoord.t = 1.0-vTextureCoord.t;
   }
   //convert the current texture coordinate to a spherical position
   vec3 imgcRad = vec3(radians(imgCenter.x),radians(imgCenter.y),imgCenter.z);
   vec3 spos = convertToSpherical(imgcRad, texCoord);
   //convert that spherical position to a cartesian coordinate
   vec3 cpos = sphericalToCartesian(spos);
   //get the distance between the pixel location and this images center
   vec3 imgc = sphericalToCartesian(imgcRad);
   float dist = distance(cpos, imgc);
   dist = .95*dist; //make this images distance slightly closer to
                    //favor pixels on this image and prevent gaps

   float ndist = 0.0;
   float alpha = 1.0;

   for (int i=0; i<8; i+=1){
       if (i < numNeighbors){
           //get the distance between neighbor center and pixel location
           ndist = distance(cpos, neighborCenters[i]);
           if (ndist < dist){
               //if any neighbor distance is less than this
               //images distance, set the alpha to 0
               alpha = 0.0;
               break;
           }
       }
   }
```

```
  vec4 textureColor =
    texture2D(uSampler, vec2(vTextureCoord.s, vTextureCoord.t));
  gl_FragColor = vec4(textureColor.rgb, textureColor.a * uAlpha * alpha);
}
```

References

[1] National Ignition Facility & Photon Science - Bringing Star Power to Earth. `http://lasers.llnl.gov/`. Accessed August 27, 2011.

[2] Laser Inertial Fusion Energy. `http://life.llnl.gov/`. Accessed August 27, 2011.

[3] R. C. Montesanti, M. A. Johnson, E. R. Mapoles, D. P. Atkinson, J. D. Hughes, and Reynolds J. L. Phase-shifting diffraction interferometer for inspecting nif ignition-target shells. Monterey, CA, United States, October 2006. American Society for Precision Engineering.

[4] C. Ware and S. Osborne. Exploration and virtual camera control in virtual three dimensional environments. *Proc. Symposium on Interactive 3D Graphics*, pages 175–183, 1990.

[5] jquery: The Write Less, Do More, JavaScript Library. `http://jquery.com/`. Accessed September 3, 2011.

[6] JavaScriptMVC. `http://javascriptmvc.com`. Accessed September 3, 2011.

[7] Java.net. `http://jersey.java.net`. Accessed September 3, 2011.

[8] Michael A. Johnson. Psdi shell geometry. Unpublished memorandum, November 2009.

[9] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *SIGGRAPH Comput. Graph.*, 22:65–74, June 1988. ISSN 0097-8930. doi: http://doi.acm.org/10.1145/378456.378484. URL `http://doi.acm.org/10.1145/378456.378484`.

[10] William E. Lorensen and Harvey E. Cline. *Marching cubes: a high resolution 3D surface construction algorithm*, pages 347–353. ACM, New York, NY, USA, 1998. ISBN 1-58113-052-X. doi: 10.1145/280811.281026. URL `http://dl.acm.org/citation.cfm?id=280811.281026`.

[11] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 303–312, New York, NY, USA, 1996. ACM. ISBN 0-89791-746-4. doi: http://doi.acm.org/10.1145/237170.237269. URL `http://doi.acm.org/10.1145/237170.237269`.

[12] Greg Turk and Marc Levoy. Zippered polygon meshes from range images. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 311–318, New York, NY, USA, 1994.

ACM. ISBN 0-89791-667-0. doi: http://doi.acm.org/10.1145/192161.192241. URL `http://doi.acm.org/10.1145/192161.192241`.

[13] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: exploring photo collections in 3d. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 835–846, New York, NY, USA, 2006. ACM. ISBN 1-59593-364-6. doi: http://doi.acm.org/10.1145/1179352.1141964. URL `http://doi.acm.org/10.1145/1179352.1141964`.

[14] Noah Snavely, Rahul Garg, Steven M. Seitz, and Richard Szeliski. Finding paths through the world's photos. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 15:1–15:11, New York, NY, USA, 2008. ACM. ISBN 978-1-4503-0112-1. doi: http://doi.acm.org/10.1145/1399504.1360614. URL `http://doi.acm.org/10.1145/1399504.1360614`.

[15] Paul Clough and Simon Read. Key design issues with visualising images using google earth. In *Proceedings of the IR research, 30th European conference on Advances in information retrieval*, ECIR'08, pages 570–574, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78645-7, 978-3-540-78645-0. URL `http://dl.acm.org/citation.cfm?id=1793274.1793345`.

[16] Sixto Ortiz Jr. Is 3d finally ready for the web? *Computer*, 43:14–16, January 2010. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/MC.2010.15. URL `http://dx.doi.org/10.1109/MC.2010.15`.

[17] Marco Di Benedetto, Federico Ponchio, Fabio Ganovelli, and Roberto Scopigno. Spidergl: a javascript 3d graphics library for next-generation www. In *Proceedings of the 15th International Conference on Web 3D Technology*, Web3D '10, pages 165–174, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0209-8. doi: http://doi.acm.org/10.1145/1836049.1836075. URL `http://doi.acm.org/10.1145/1836049.1836075`.

[18] Benjamin P. DeLillo. Webglu development library for webgl. In *ACM SIGGRAPH 2010 Posters*, SIGGRAPH '10, pages 135:1–135:1, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0393-4. doi: http://doi.acm.org/10.1145/1836845.1836989. URL `http://doi.acm.org/10.1145/1836845.1836989`.

[19] Catherine Leung and Andor Salga. Enabling webgl. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 1369–1370, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. doi: http://doi.acm.org/10.1145/1772690.1772933. URL `http://doi.acm.org/10.1145/1772690.1772933`.

[20] Marco Callieri, Raluca Mihaela Andrei, Marco Di Benedetto, Monica Zoppè, and Roberto Scopigno. Visualization methods for molecular studies on the

web platform. In *Proceedings of the 15th International Conference on Web 3D Technology*, Web3D '10, pages 117–126, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0209-8. doi: http://doi.acm.org/10.1145/1836049.1836067. URL `http://doi.acm.org/10.1145/1836049.1836067`.

[21] Michael Kazhdan, Dinoj Surendran, and Hugues Hoppe. Distributed gradient-domain processing of planar and spherical images. *ACM Trans. Graph.*, 29:14:1–14:11, April 2010. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1731047.1731052. URL `http://doi.acm.org/10.1145/1731047.1731052`.